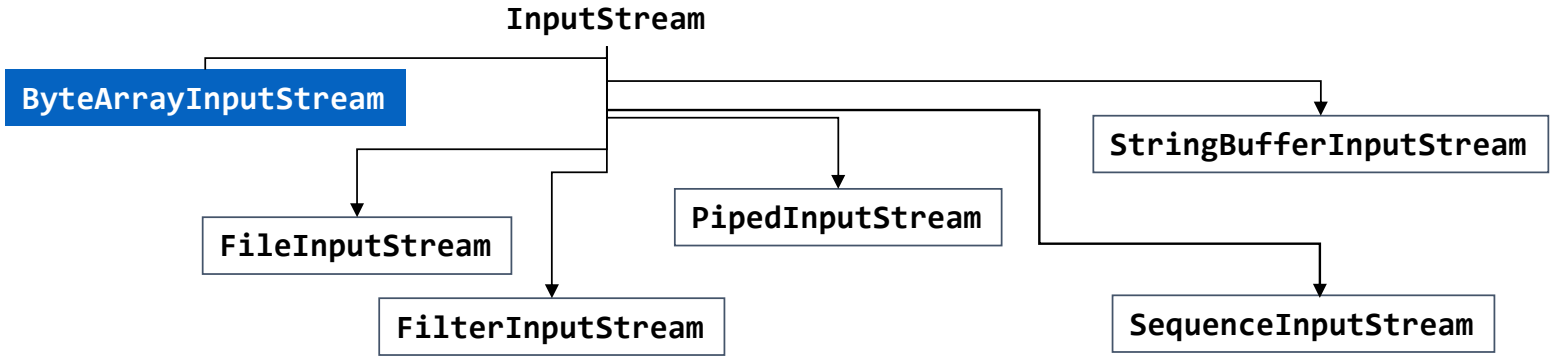
Etape 3

- Ensemble des Flux Binaires

L'API JAVA incorpore beaucoup d'autres flux afin de prendre en compte des sources et destinations particulières ou des prétraitements facilitant la manipulation des données par les programmes

ByteArrayInputStream



- *ByteArray[Input/Output]Stream* : possède un buffer interne (passé en paramètre du constructeur) qui peut être lu à partir du flux

[sorte de tableau en mémoire qui sert de source de données]

- Un compteur interne garde une trace du prochain octet à lire. Une des particularités est que la méthode close() n'a aucun effet.

- *Constructeurs* :

`ByteArrayInputStream(byte[] buf)`: création d'un *ByteArrayInputStream*

ByteArrayInputStream

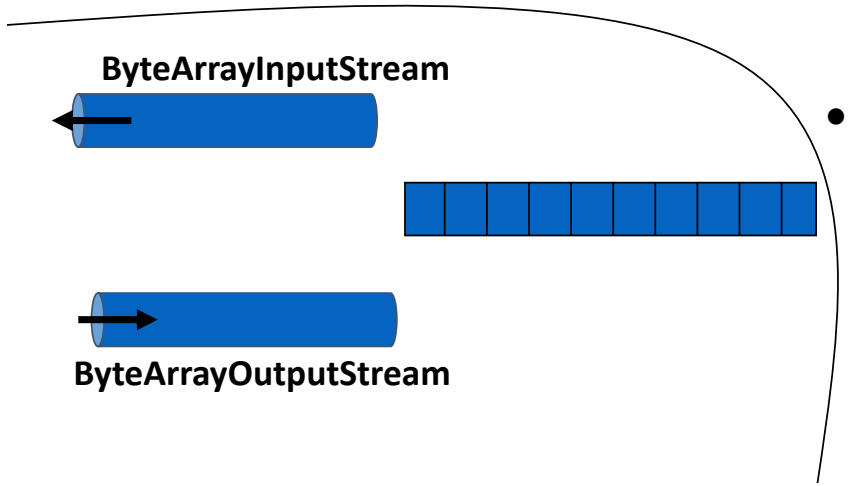
- *Constructeurs* :

ByteArrayInputStream(byte[] buf): création d'un ByteArrayInputStream

- *Constructeurs du flux en écriture* :

ByteArrayOutputStream():

The buffer capacity is initially 32 bytes, though its size increases if necessary



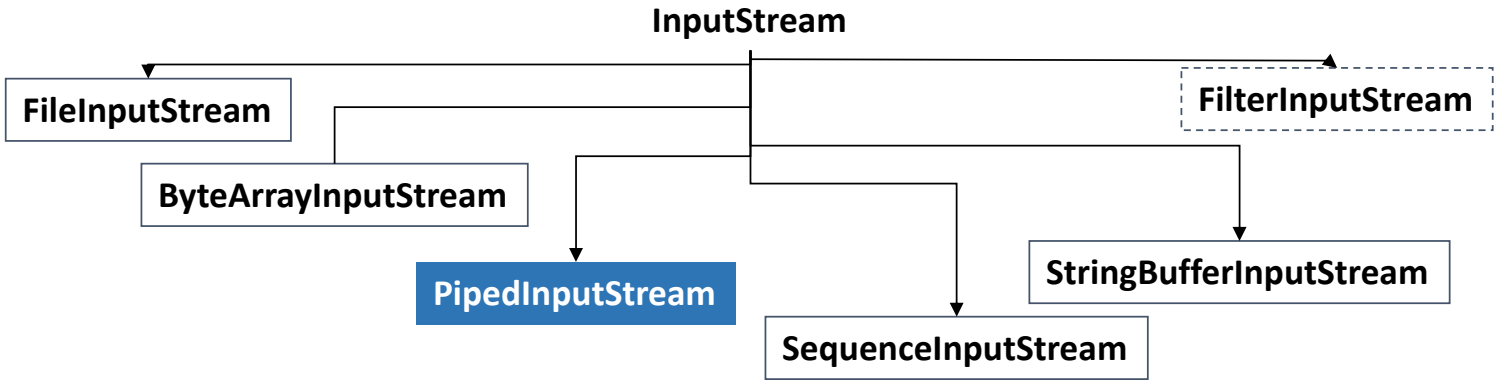
- **Constructor arguments**

The buffer from which to extract the bytes.

- **How to use it**

As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.

PipedInputStream



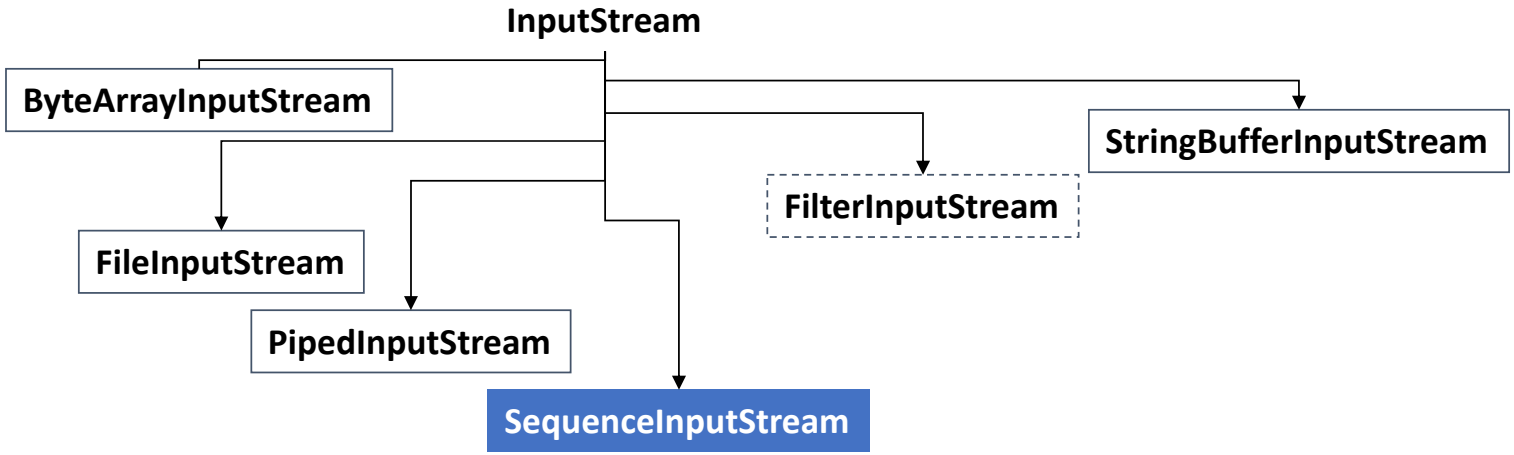
- *Piped[Input/Output]Stream* : cette classe est largement inspirée du PIPE UNIX (la sortie d'une instruction correspond à l'entrée d'une autre instruction).
- Elle travaille de paire avec un *PipedOutputStream* auquel elle est connectée. Typiquement, les données sont lues à partir d'un PipedInputStream par un thread et écrites sur le PipedOutputStream par un autre thread.
- La connexion est effectuée à la création de l'objet (via le constructeur) ou par la suite grâce à la méthode connect().
  - *Constructeurs et méthodes (les symétriques existent pour PipedOutputStream):*

PipedInputStream(PipedOutputStream src)
Creates a PipedInputStream so that it is connected to the piped output stream src.

PipedInputStream()
Creates a PipedInputStream so that it is not yet connected.

connect(PipedOutputStream src)
Causes this piped input stream to be connected to the piped output stream src.

SequenceInputStream



- ***SequenceInputStream :*** représente la concaténation logique de plusieurs flux en lecture. Lorsqu'on atteint la fin d'un flux, la lecture continue sur le flux suivant. Les flux de lecture sont passés au constructeur sous forme d'***Enumeration***

- *Constructeurs :*

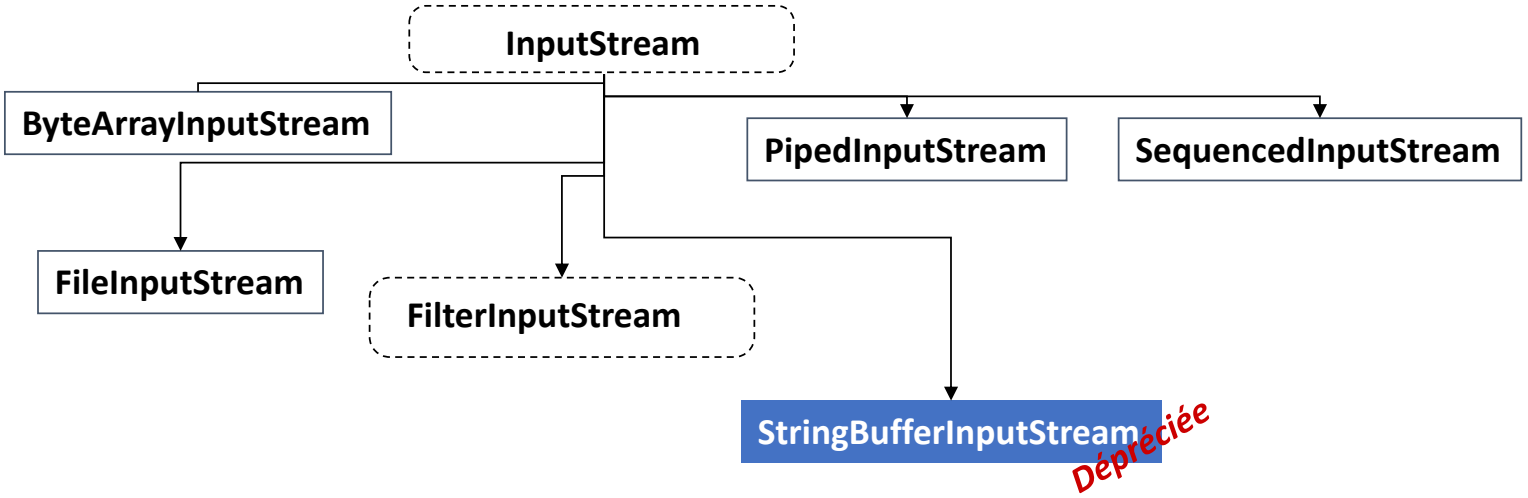
SequenceInputStream(Enumeration e)

Initializes a newly created SequenceInputStream by remembering the argument, which must be an Enumeration that produces objects whose run-time type is InputStream.

SequenceInputStream(InputStream s1, InputStream s2)

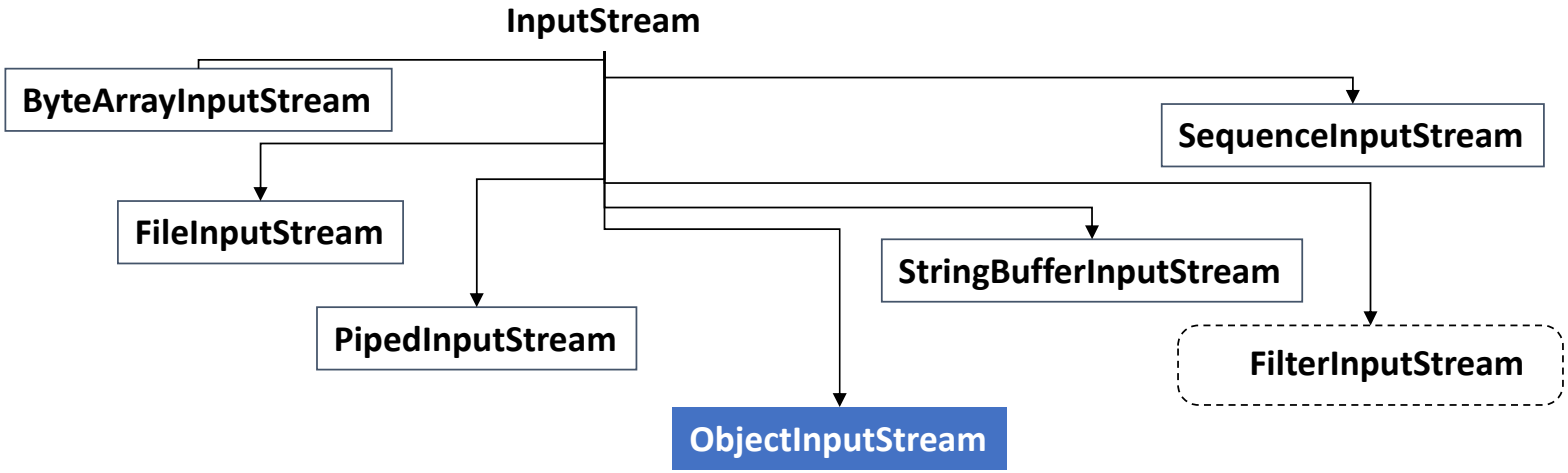
Initializes a newly created SequenceInputStream by remembering the two arguments, which will be read in order, first s1 and then s2, to provide the bytes to be read from this SequenceInputStream.

StringBufferInputStream



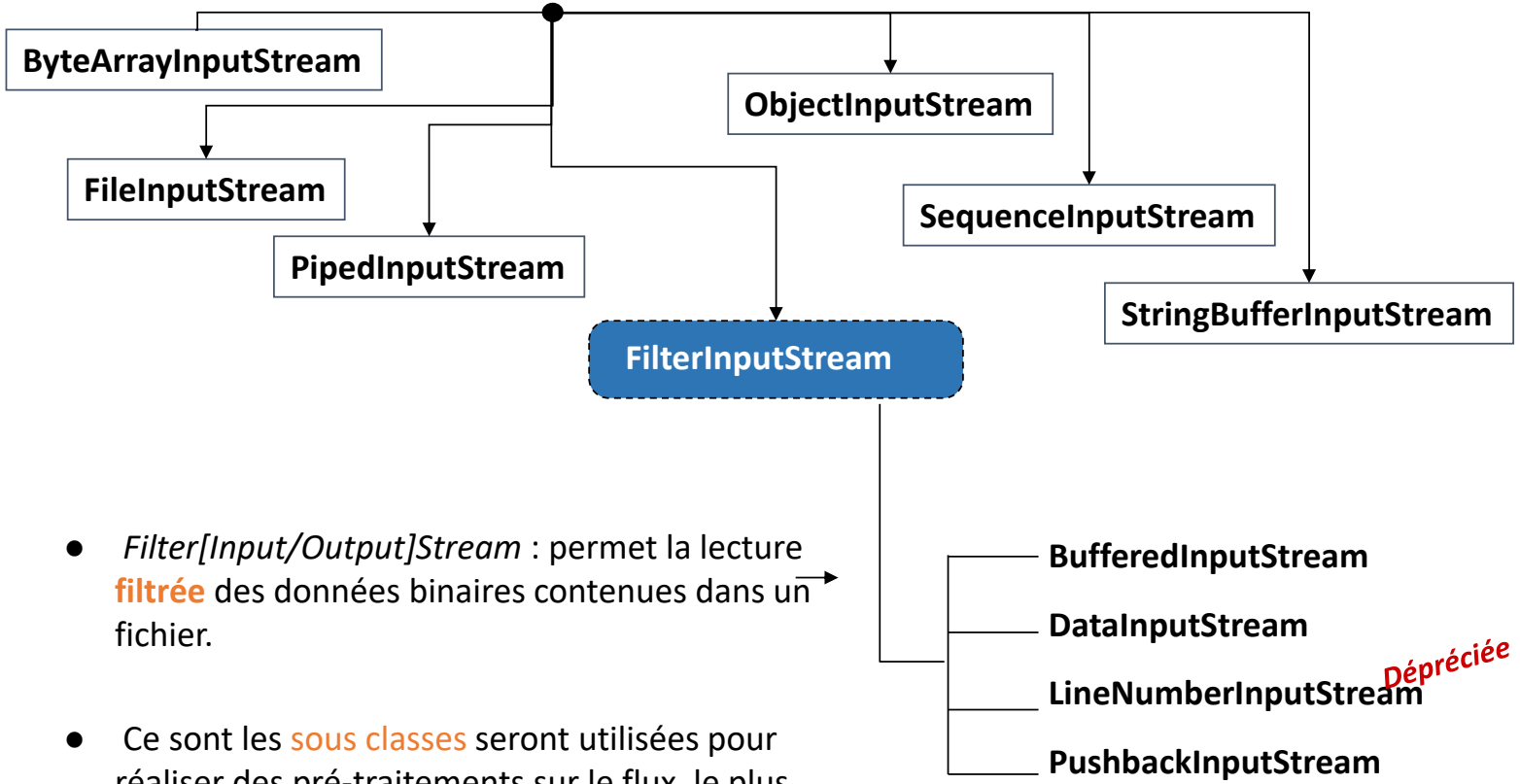
- *StringBuffer [Input/Output]Stream* : cette classe (dépréciée) connecte le flux à une chaine de caractère. Utiliser plutôt *Streamtokenizer ou StringReader*

ObjectInputStream



- *Object[Input/Output]Stream* : cette classe sert pour la désérialisation.
  Elle possède différentes méthodes pour lire tout type de données (primitifs ou objets).

FilterInputStream



- *Filter[Input/Output]Stream* : permet la lecture **filtrée** des données binaires contenues dans un fichier.

- Ce sont les sous classes seront utilisées pour réaliser des pré-traitements sur le flux, le plus simple étant la mise en tampon.

Les flux de traitements (ou flux filtrés)

BufferedInputStream

FLUX BUFFERISES

- informations envoyées sous la forme de gros volumes (plus rapide) ;
- principe du flux à tampon (exemple : recevoir un livre page/page ou chapitre par chapitres) :
  - Le flux à tampon remplit le tampon de données qui n'ont pas encore été traitées ;
  - quand un programme a besoin de ces données il scrute le tampon ;

- Constructeurs :

```
BufferedIntputStream(InputStream)
BufferedIntputStream(InputStream , taille)
```

  - InputStream : la source
  - taille=taille du tampon.

Exemple :

```
BufferedInputStream bis = new BufferedInputStream(System.in);
```

- Les données du tampon ne sont transférées à leur destination que lorsqu'il est rempli ou que la méthode *flush()* a été appelée sur le tampon.

Les classes des flux filtrés : FilterInputStream

- **DataInputStream** : formatage des données lues ou écrites. Offre des fonctionnalités de lecture de données qui sont des types primitifs JAVA dans un flux binaire entrant :

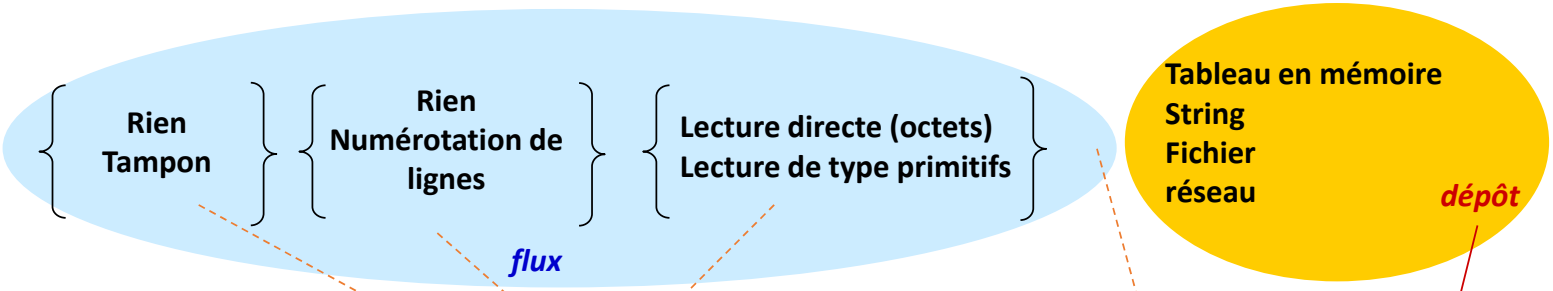
boolean readBoolean(); char readChar(), int readInt(), float readFloat(), …

Réservé à une lecture / écriture de ce type : un fichier écrit avec ce type de flux ne pourra être lu qu'avec ce même type.

- **LineNumberInputStream** : fournit le numéro de la ligne courante lue.
- **PushBackInputStream** : Ajoute la possibilité de revenir en arrière pour relire des octets . **dépréciée**

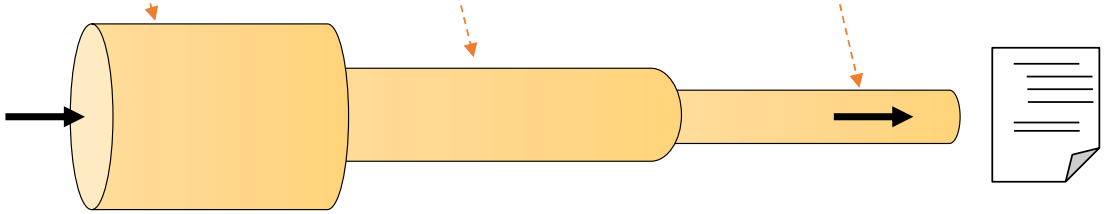
Elaboration des flux de traitement

Constitution des flux composites : « emboîtement » successif (en entrée comme en sortie)



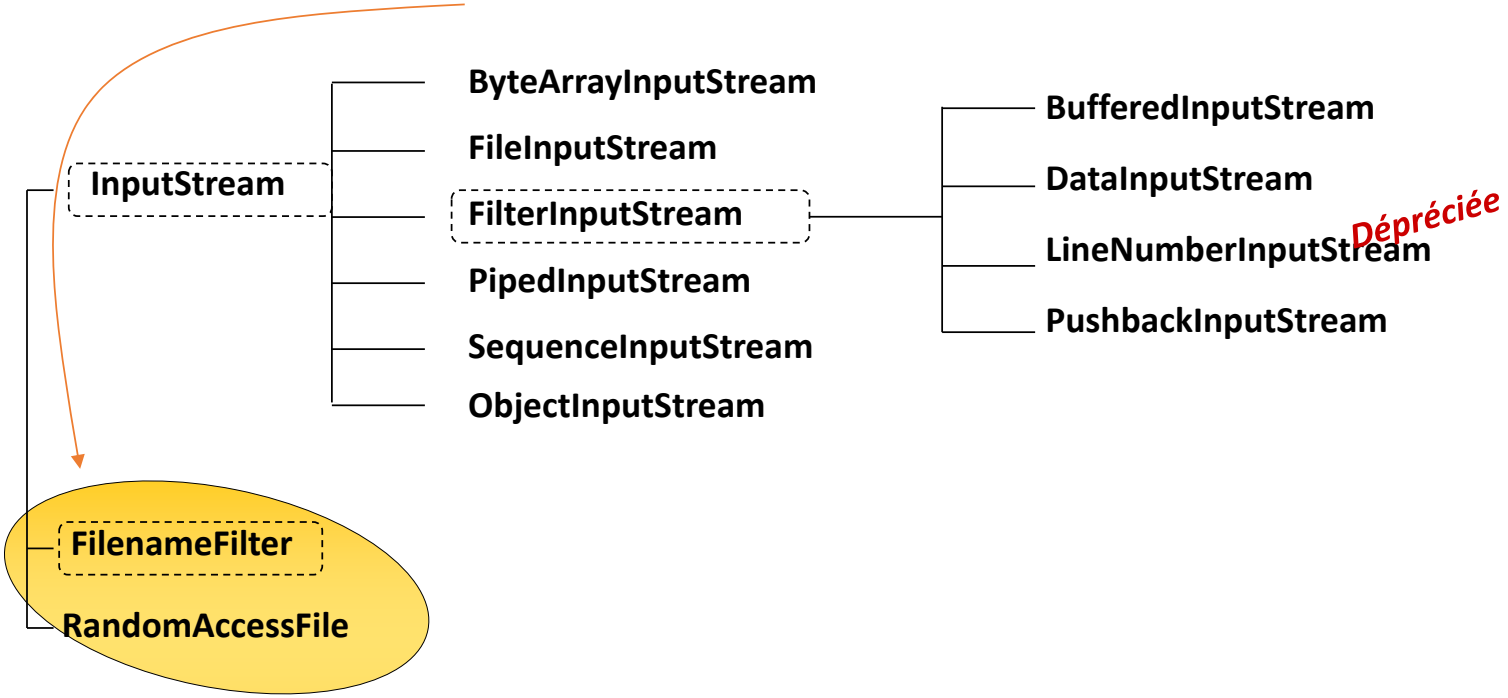
Ouverture d'un flux binaire sortant tamporisé sur un fichier

```
DataOutputStream out = null;
try
{
out = new DataInputStream(new BufferedInputStream(new FileInputStream(« NomDuFichier »)));
}
catch ….
```



```
StreamTokenizer lec= new StreamTokenizer( new InputStreamReader(System.in));
```

Autres classes des flux binaires

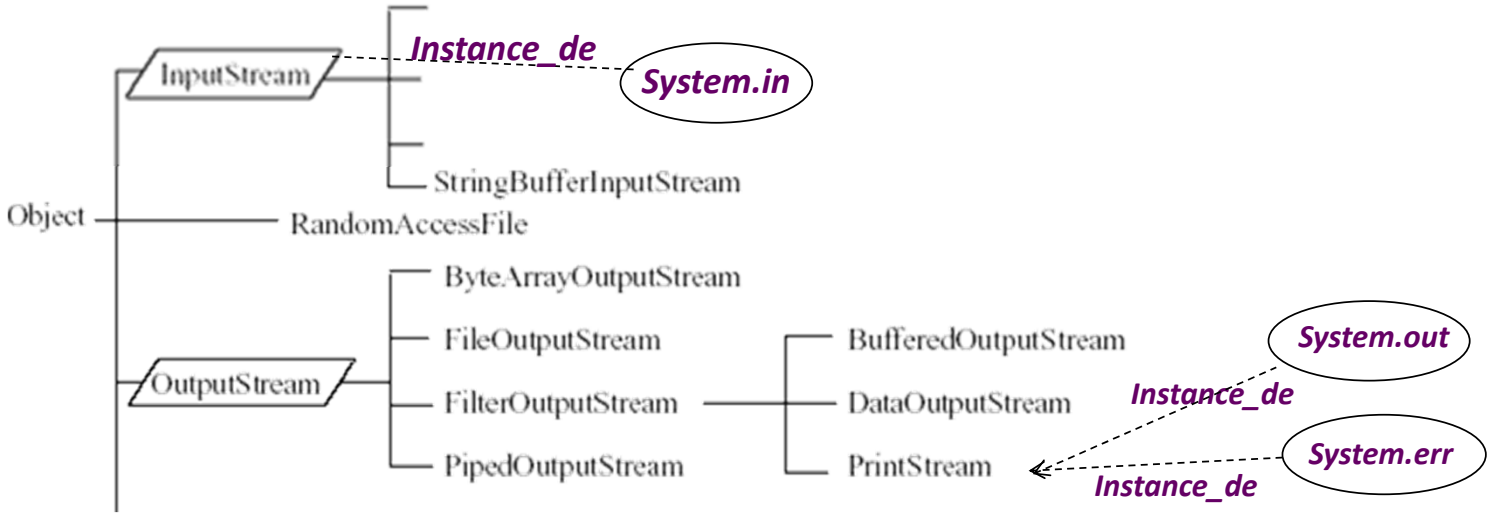


- La classe *RandomAccesFile* supporte la lecture et l'écriture de manière directe (et non séquentiellement) vers un fichier à accès aléatoire. Ce dernier se comporte comme un grand tableau d'octets stocké dans le système de fichiers.

L'interface *FilenameFilter* sert de filtre sur les noms de fichier. Elle possède la méthode (à redéfinir dans l'implémentation) *public boolean accept(File dir, String name)* qui sert à filtrer le contenu d'un répertoire. Celle-ci renvoie *true* si le fichier appelé « *name* » peut-être inclus dans une liste de fichier.

PrintStream et quelques flux prédéfinis

- 3 flux prédéfinis
  - **static System.in** : instance de la classe ***InputStream*** : l'entrée standard ;
  - **static System.out** :instance de la classe ***PrintStream*** : la sortie standard ;
  - **static System.err** : instance de la classe ***PrintStream*** : la sortie standard d'erreurs.



PrintStream adds functionality to another output stream :

- ability to **print representations of various data values conveniently**.
- a PrintStream **never throws an IOException**; instead, exceptional situations merely set an internal flag that can be tested via the checkError method.
- a PrintStream can be created so as to **flush automatically**; this means that the flush method is automatically invoked after a byte array is written, one of the println methods is invoked, or a newline character or byte ('\n') is written.

- Flux binaires pour la création d'archives compressées

1.3 Compression de fichiers

- Les classes spécifiques à la compression ZIP sont issues du package *java.util.zip*.

- Le JRE fournit également d'autres classes permettant de compresser dans d'autres formats (GZIP et JAR) dont les principes restent les mêmes.

- La classe *ZipOutputStream* permet la compression des données via sa méthode *write(byte[] b, int off, int len)*.
  - Elle est associée à un flux de communication sur lequel on lit les données compressées (passé en paramètre au constructeur).

- En Java, une archive ZIP est composée d'un ou plusieurs *ZipEntry*.
  = items correspondant à des fichiers (ou répertoire) compressés ⇒ un *ZipEntry* pour chaque item.

Exemple de compression

à importer

```
import java.io.*;
import java.util.zip.*;
```

```
int BUFFER = 2048;
try
        { //On crée le fichier
         FileOutputStream fos = new FileOutputStream "C:\\dev\\archive.zip");
    //On crée le flux de compression
         ZipOutputStream zipos = new ZipOutputStream(fos);
         byte[] data = new byte[BUFFER];

         //On ouvre le flux sur le fichier à lire
         FileInputStream fis =new FileInputStream("C:\\fichier.txt");

         //On crée un item correspondant au fichier d'origine
         ZipEntry ze = new ZipEntry("fichier.txt");

         //On place l'item dans le ZIP
         zipos.putNextEntry(ze);
```

Constructeur : nom qu'aura **l'item** dans **l'archive** (généralement le nom du fichier)

L'item est ensuite ajouté au *ZipOutputStream* grâce à la méthode *putNextEntry(ZipEntry e)*

Exemple de compression

```
      int count;
//Tant qu'on peut lire
      while ((count = fis.read(data, 0, BUFFER)) != -1)
          {
     zipos.write(data, 0, count);
          }


//On ferme l'item dans le zip
     zipos.closeEntry();

     //On ferme le flux vers le ZIP
     zipos.close();
      }           catch (Exception e) { e.printStackTrace(); }    }   }
```

Toutes les données écrites sur le flux seront alors associées à cet item jusqu'à l'appel de la méthode *closeEntry()*

Exemple de compression

```
try
 {
  //On crée le flux de communication de lecture
  FileInputStream fis = new FileInputStream("C:\\dev\archive.zip");

  //On crée le flux de décompression
  ZipInputStream zis = new ZipInputStream(fis);
     //On récupère chaque item
  ZipEntry entry;
  while ((entry = zis.getNextEntry()) != null)
     {
     int count;
     byte[] data = new byte[BUFFER]; // constante de 2048

  //On récupère chaque item
 FileOutputStream fos = new FileOutputStream("C:\\dezip\\«  + entry.getName());
  //Tant que l'item contient des données
     while ((count = zis.read(data, 0, BUFFER)) != -1)
          {
```

- Récupération de l'item suivant dans l'archive (le ZipEntry)

- La décompression s'effectue grâce à la méthode *read(byte[], int off, int len)* de la classe *ZipInputStream*
- *read* lit les données compressées d'un *ZipEntry*

- Pour savoir à quel fichier d'origine appartiennent les données, il suffit de faire appel à la méthode *getName()* du *ZipEntry*.